Bogor: An Extensible and Highly-Modular Software Model Checking Framework*

Robby
Department of CIS
Kansas State University
robby@cis.ksu.edu

Matthew B. Dwyer
Department of CIS
Kansas State University
dwyer@cis.ksu.edu

John Hatcliff
Department of CIS
Kansas State University
hatcliff@cis.ksu.edu

ABSTRACT

Model checking is emerging as a popular technology for reasoning about behavior properties of a wide variety of software artifacts including: requirements models, architectural descriptions, designs, implementations, and process models. The complexity of model checking is well-known, yet cost-effective analyses have been achieved by exploiting, for example, naturally occurring abstractions and semantic properties of a target software artifact. Adapting a model checking tool to exploit this kind of *domain knowledge* often requires in-depth knowledge of the tool's implementation.

We believe that with appropriate tool support, domain experts will be able to develop efficient model checking-based analyses for a variety of software-related models. To explore this hypothesis, we have developed Bogor, a model checking framework with an extensible input language for defining domain-specific constructs and a modular interface design to ease the optimization of domain-specific state-space encodings, reductions and search algorithms. We present the pattern-oriented design of Bogor and discuss our experiences adapting it to efficiently model check Java programs and event-driven component-based designs.

1. INTRODUCTION

The development of modern software systems involves a rich complex of software artifacts. Artifacts may represent aspects of the process by which software is developed or the various products that are outcomes of stages of process phases. Disciplined approaches to software development will apply criteria to determine essential properties of such artifacts, for example, a requirements model should be checked for completeness and consistency prior to commencement of system design. For large complex software systems such criteria should be formalized and their checks automated. While reasoning about structural properties of artifacts is efficient and can be useful (e.g., interface signatures, connection conformity), behavioral reasoning is the ultimate goal.

Temporal logic model checking [10] is a powerful framework for reasoning about the behavior of finite-state system descriptions and it has been applied, in various forms, to reasoning about a wide-variety of software artifacts. For example, model checking frameworks have been applied to reason about software process models, (e.g., [29, 31]), different families of software requirements models (e.g., [3, 8]), architectural frameworks (e.g., [21, 30]), design models, (e.g., [1, 24]), and system implementations (e.g., [5, 11, 22]). The effectiveness of these efforts has in most cases relied on detailed knowledge of the model checking framework being applied. In some cases, a *new* framework was developed targeted to the semantics of a family of artifacts [5, 22], while in other cases it was necessary to study an *existing* model checking framework in detail in order to customize it [8, 13]. Unfortunately, this level of knowledge and effort currently prevents many domain experts from successfully applying model checking to software analysis. Experts in different areas of software engineering have significant domain knowledge about the semantic primitives and properties of families of artifacts that could be brought to bear to produce cost-effective semantic reasoning via model checking. However, in order to leverage this domain knowledge in model-checking, these domain experts should not be required to build their own model-checker or to pour over the details of an existing model-checker implementation while carrying out substantial modifications.

In this paper, we describe a new model checking framework called *Bogor* that is designed to make it easy to extend the model checker with new semantic primitives and to optimize the state-space storage and exploration strategies for particular domains. Existing model checkers, such as SPIN [26], FDR2 [19], and NuSMV [9], were designed to support a fixed input language using a fixed collection of state-space representations, reduction and exploration algorithms. The capabilities of these tools has evolved over time, but that evolution has been limited to the capabilities that the tool developer found useful or desirable. While such model checking frameworks are in widespread use, there are multiple difficulties in applying them to reason about a broad range of software artifacts. Below we describe in greater detail the challenges of working with existing model-checkers and how Bogor addresses these challenges.

The input language gap: Semantic primitives used in defining software artifacts cover a broad range from typical arithmetic and logical operations, which are well-supported

*This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by Rockwell-Collins and by Intel Corporation (Grant 11462).

by existing model checkers, to domain-specific large-grain operations [21, 24]. It is not always possible to map domain constructs efficiently onto model checker input languages. For example, most model checking frameworks do not support systems whose state evolves dynamically (e.g., dynamic creation of data or threads of control) and the few systems that have attempted to address these kinds of features have built special purpose model checkers [5, 13] or settled for bounded static approximations of dynamic behavior [11]. *Analysis developers should be able to define domain-specific primitives to adapt the model checker to a family of software artifacts.*

Property language overhead: Model checking frameworks typically come with the ability to check implicit properties, such as deadlock, system-specific invariants, and temporal logic formulae. It is often the case, however, that reasoning about a family of software artifacts requires a property language, for example safety properties, that may be much simpler to check than a full temporal logic. Furthermore, encoding properties with the limited features of model checker expression languages can require the addition of state variables [12] that unnecessarily expand the state space. *Analysis developers should be able to customize the property checking component of a model checker to optimize its performance for targeted properties.*

Fixed state encodings: Most model checking frameworks, with the exception of [6], employ a small fixed set of strategies for encoding system data. Knowledge about domain-specific data, components and properties may suggest opportunities for particularly efficient encodings that cannot be easily incorporated in existing model checking frameworks. *Analysis developers should be free to tailor the encoding of system data to achieve state space reductions.*

Fixed search algorithms: Model checking typically involves a stateful search of a system's reachable state-space. For defect detection one may perform a less complete heuristic search [18, 23, 32] and for certain classes of systems stateless search may be an attractive alternative [22]. *Analysis developers should be able to configure the search mode of the model checker based on the kinds of reasoning desired and on the nature of the artifact.*

Fixed reduction strategies: There exists an enormous body of literature on state space reduction strategies. For example, partial order reductions which exploit information about independence between transitions (e.g., [10, Chapter 10]) and abstraction-driven-refinement (e.g., [25]) are techniques that have proven quite effective in reducing model checking costs. Interestingly, the theoretical presentation of these techniques is often parameterized by, e.g., the independence relations or counter-example feasibility analyses, but actual implementations are usually hard-wired to a particular notion of independence or a particular notion of feasibility. However, parameterizing the implementation on particular strategies or going beyond this to allow the strategies to be changed dynamically during checking (e.g., to dynamically calculate independence information [15]) can lead to significant improvements in reduction techniques. *Analysis developers should be able to combine collections of reductions to target a specific family of software artifacts and thereby maximize state-space reduction without incurring unnecessary overhead.*

Bogor has grown out of our significant experience developing model checking frameworks that *translate* software artifacts to the input languages of existing model checking tools. Two such frameworks are Bandera [11], which extracts abstract finite-state models Java programs and produces input for model checkers such as SPIN, and Cadena [24], which translates extended CORBA Component Model (CCM) designs of event-driven systems to the input language of the dSPIN model checker. In both of these systems we found that the deficiencies, highlighted above, of existing model checkers forced us to produce models whose analysis was less-efficient than we knew was possible based on our understanding of the two domains. Furthermore, our experience suggests that no fixed input language or collection of capabilities would suffice, since it is not possible to anticipate all of the kinds of artifact analyses that developers might want.

We have also been inspired by our experiences working with Java Path Finder (JPF) [5] – a model-checker that works directly on Java byte-code. JPF stands in contrast to many existing model-checkers in that it has proven to be a very flexible framework for incorporating a wide variety of state space exploration algorithms. Thus, JPF provides flexibility along several axes (flexibility in search algorithms, flexibility in reduction strategies), but it is strongly tied to byte-code (i.e., it is not flexible with respect to input language) and is thus not well-suited for analyzing other kinds of software artifacts.

These experiences have led us to develop a novel model checking framework that is explicitly designed to support the analysis of a wide-variety of software artifacts related to modern, dynamic, concurrent software systems. This paper makes the following specific contributions:

1. we describe the design of Bogor's *extension* interface that allows the model checkers input language to be extended to form a *virtual machine* that is adapted to different families of software artifacts;
2. we describe how Bogor's *module* and *plug-in* interfaces are engineered using design patterns that encapsulate and reduce dependencies between core modules and that allow the components of the model checking engine to be customized to a family of software artifacts;
3. we illustrate how Bogor can be customized to efficiently support checking properties of two families of software artifacts: Java programs and CCM-based designs from Cadena;
4. we demonstrate that even though Bogor is designed for flexibility, this flexibility need not degrade performance – in particular, domain-specific optimizations that can be incorporated through Bogor's flexibility yield performance numbers that often dramatically improve upon the performance of existing non-customized model-checkers.

Bogor is implemented as an Eclipse plug-in [17] and comes with an integrated editor and counter-example visualization component; for more information visit the Bogor web-site [33].

The paper proceeds in the next section with a detailed description of Bogor's input language and presents the architecture of the model checking framework. Section 3 presents a detailed example illustrating how Bogor can be extended. Section 4 gives an overview of how Bogor is being used in the Bandera [11] and Cadena [24] model checking tool-sets. Section 5 discusses related work and we conclude in Section 6.

2. BOGOR OVERVIEW

2.1 BIR Modeling Language

Bogor checks systems specified in a revised version of the Bandera Intermediate Representation (BIR). The previous version of BIR was designed to be an intermediate language used by Bandera for translating Java programs to the input languages of existing model-checkers such as Spin. Thus, this earlier version provided direct support for modeling Java features such as threads, Java locks (supporting wait/notify), and a bounded form of heap allocation. To facilitate the construction of translators to back-end model-checkers like Spin, BIR control-flow and actions are stated in a *guarded command* format which it quite close to the format used to specify systems in model-checker input languages like Promela.

As explained in the introduction, our experience with Bandera and other tools such as JPF and dSpin has led us to conclude that software model-checking can be more effectively supported by a new infrastructure that has at its core an extensible model-checker that is designed to support software directly rather than relying on translations to model-checkers that do not provide direct support for modeling many of the language features found in modern software. As part of this transition to a new infrastructure, we have revised the definition of BIR to include a number of new features such as the BIR extension mechanism, generic types and polymorphic functions, type-safe function pointers, virtual function/method tables, and unbounded dynamic allocation of heap objects and threads supported by garbage collection.

Here are some language design goals that drove our revision of BIR.

1. Provide support for modeling of a variety of software artifacts including code (e.g., Java, C#), designs (e.g., state-charts, high-level transition systems), and abstractions of software layers (e.g., CORBA middleware services and asynchronous communication mechanisms).
2. Provide type-safe modeling that supports OO type-structures and makes a clear distinction between side-effecting and non-side-effecting constructs.
3. Design the language to serve as the target of automatic translations.
4. Provide support for capturing a variety of property specification forms (e.g., temporal logic, automata, assertions, invariants) used in software model-checking.
5. Provide mechanisms that (a) support abstraction along multiple dimensions and (b) allow state data that is irrelevant to the property being verified to be shifted out of the model-checker's state-storage mechanisms.
6. Provide support for user-defined type-safe extensions of the language to include new types, expressions, and actions (commands).

The BIR data model: There are two categories of BIR data types: *primitive* types (e.g., `boolean`, `int`, `long`, `float`, `double`, `enum`), and *non-primitive* types (e.g., `null`, `record`, array, `lock`). BIR's type system is designed to *support* modeling of Java programs but not to be hard-wired to the Java type system (design Goal 1 above). For example, BIR does not include the Java notion of 'class' directly, but instead models class data and virtual methods using records and type safe function pointers. This flexibility makes it easier to model structures found in other software artifacts such as the high-level component-based designs from Cadena [24]. Java's inheritance-based sub-typing and virtual methods are supported by explicit BIR sub-typing declarations on record types and by virtual method tables which are included as primitives in BIR. In addition, BIR adopts the Java memory model (i.e., pointer arithmetic is disallowed and object reclamation is achieved through garbage collection). Although this appears to go against Goal 1, it is necessary to achieve our goals for safety (Goal 2). Moreover, most applications domains that seek to apply model-checking to achieve high-assurance disallow pointer arithmetic anyway.

BIR emphasizes strong static typing (Goal 2) for several reasons. Strong static typing minimizes the amount of dynamic type checking that the model checker needs to perform, thus, it reduces overhead during model-checking. Additionally, strong static typing provides earlier feedback concerning the well-formedness of models, and this aids in the correct design of translators targeting BIR and manual model construction.

For flexibility, BIR supports generic abstract data types (ADTs) and polymorphically typed functions. However, Bogor does not currently provide polymorphic type-inference as in SML or Haskell. Instead, type variables for functions and ADTs must be instantiated explicitly. We have chosen this approach because BIR automatic translation can easily supply explicit polymorphic types and instantiation (Goal 3), and because sophisticated polymorphic type inference for OO-like type systems includes a number of non-trivial issues that are peripheral to our central theme of model-checking.

In contrast to the input languages used in almost all other model-checkers (including SPIN [26], NuSMV [9], SLAM [2], BLAST [25]), BIR supports dynamic creation of both thread and heap objects with automatic reclamation by garbage collection. Moreover, BIR provides a state-of-the-art canonical heap representation that seems essential for effective checking of highly dynamic concurrent software systems. Such systems generate many heap instances that differ in the relative position of allocated objects but that are actually *observationally equivalent* (i.e., the heap instances can not be distinguished by any Java memory operations). Due to positional differences of object placement in heaps, conventional representations of heaps as e.g., arrays of memory cells would yield different states for these heaps. However, Bogor's canonical heap representation (based on work by Iosif on dSpin [13]) ensures that heaps that are observationally equivalent map to the same state. This dramatically reduces the number of states generated when checking highly dynamic systems.

The BIR control model: BIR's guarded commands are formed using two classes of constructs: guard *expressions* which evaluate to values but produce no side-effects, and *actions* (commands) which modify the system's data state (Goal 2). The rich expression language includes function expressions and value bindings as found in functional programming languages. BIR provides exception handling mechanisms that would be otherwise tedious to model using the traditional control-flow jump instructions (Goal 1).

BIR Extensions: BIR's extension facility allows modelers to add new types, expressions, and actions (Goal 6). An extension declaration consists of (1) a signature declaration which specifies the new symbols and associated arities to be introduced into the name-spaces for types, expressions, and actions, and (2) the name of a Java package that implements

```
system ResourceContention {
  extension Set for myPackage.SetModule {
    typedef type<'a>;
    expdef Set.type<'a> create<'a>('a ...);
    expdef 'a choose<'a>(Set.type<'a>);
    expdef boolean isEmpty<'a>(Set.type<'a>);
    expdef boolean forAll<'a>('a -> boolean,
                              Set.type<'a>);
    actiondef add<'a>(Set.type<'a>, 'a);
    actiondef remove<'a>(Set.type<'a>, 'a);
  }

  record Resource { boolean isFree; }
  record Disk extends Resource { }
  record Display extends Resource { }

  Set.type<Resource> resourcePool;

  fun isResourceFree(Resource resource)
      returns boolean = resource.isFree;

  fun AreAllResourcesInPoolFreeInv()
      returns boolean =
    Set.forAll<Resource>(isResourceFree,
                         resourcePool);

  main thread MAIN() {
    loc loc0:
      do { // create the pool and creates two processes
        resourcePool := Set.create<Resource>
            (new Disk, new Disk, new Display);
        start Process(); start Process();
      } return;
  }

  thread Process() {
    loc loc1:
      invoke run()
      return;
  }

  function run() {
    Resource resource;
    loc loc2:
      when !Set.isEmpty<Resource>(resources)
      do { // choose an element and remove it
        resource := Set.choose<Resource>
            (resourcePool);
        Set.remove<Resource>(resourcePool,
                             resource);
      } goto loc3;
    loc loc3:
      do { // resource in use
        resource.isFree := false;
      } goto loc4;
    loc loc4:
      do { // resource free
        resource.isFree := true;
      } goto loc5;
    loc loc5:
      do { // add the resource back to pool
        Set.add<Resource>(resourcePool,
                          resource);
      } goto loc2;
      do { // empty transformation
      } goto loc2;
  }
}
```

Figure 1: Resource Contention Example

the semantics of the extension. Note that the extension does not extend the BIR grammar, but only adds names to the set of names of built-in expressions, actions, etc. This means that the developer does not need to extend the parser or other syntactic support facilities. Rather, the developer traverses the extension structure and implements the extension semantics using well-defined APIs for BIR's existing abstract syntax trees and Bogor model-checker components.

Extensions provide a convinient way to realize domain-specific abstractions of software components or layers, and to address the input language gap for modeling domain-specific software artifacts (Goal 5). Often times, there are component/layers of the software that have a significant amount of state that is *irrelevant* to the properties being checked. Rather than maintaining a complete implementation of a software component/layer using BIR's variables, the Java package implementing the extension can hold the state associated with the complex component/layer and only expose as much as is relevant at the BIR level. Since only the BIR variables are held in Bogor's state vector during model-checking, this can dramatically reduce the costs of representing portions of a software system.

Hiding complex portions of the state in this manner is not novel. For example, when modeling communication protocols using Spin [26], channel data types (`chan`) are often used to model message-passing channel in networks. This can be done because the specific implementation of the network channels is *irrelevant* with respect to the properties being checked. The properties are usually only concerned with the functional behavior of the channels, i.e., rendevouz, asynchronous, synchronous, and sending and receiving messages. Furthermore, properties are usually only concerned with a channel's *abstract* states, i.e., the specific channel implementation state, for example, the state of message retransmission protocols used to provide the channel service, does not need to be exposed in the model state. What is novel about BIR is that it does not a priori hard-code such abstraction mechanisms for a particular domain – rather, BIR's extension facility provides an open-ended mechanism for adding any number of domain-specific abstractions. For example, BIR extensions are used to represent the functionality of the Real-Time CORBA Event Service in our work on checking CCM designs [24].

BIR Example: Figure 1 presents a BIR model of concurrent processes that acquire and release resources from a resource pool. The code begins with an extension declaration of a generic set type used to represent the set of resources. We will show the implementation details of the `SetModule` as a Bogor plug-in in Section 3, and for now simply comment on aspects of its signature. The new type variable `'a` represents the element type of the set. The extension includes both expression extensions (create, choose, isEmpty, forAll) and action extensions (add, remove). The create expression creates and returns a new set (this is considered non-side-effecting since it does not modify any BIR variables). The ellipses (...) in the argument of choose indicate that it has variable arity. The choose expression non-deterministically picks one element of the argument set to return (the non-determinism is implemented internally by calling a Bogor primitive that causes the model-checker scheduler to explore multiple paths that correspond to the return every element of the argument set). The `forAll` expression takes a predicate (represented as a boolean function) and a set and determines whether all elements of the set satisfy the predicate. In general, an expression extension may perform complex computations atomically and it can use any information available to the model checker such as the state in which it is currently being. However, it must not modify the stored states of the model-checker. In general, an action extension can do whatever an expression can do (except return

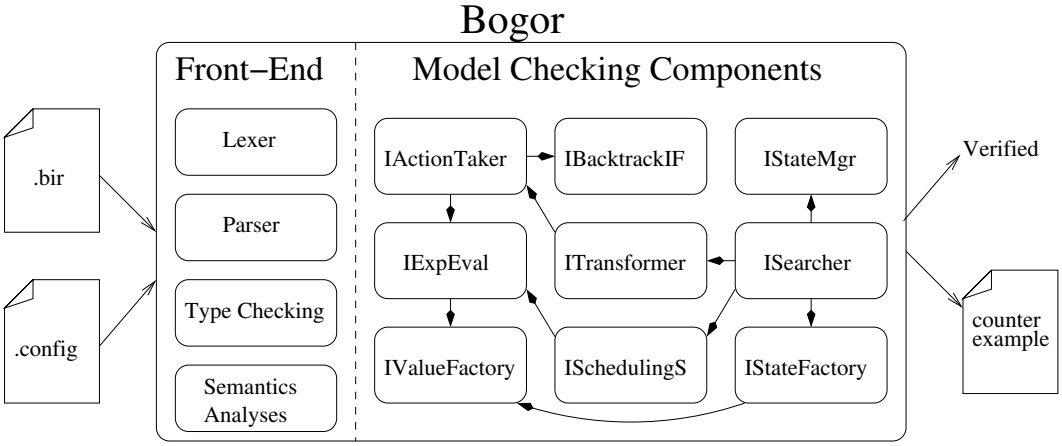

Figure 2: Bogor Architecture

a value), and it can also modify the current state.

A record type declaration begins with a declaration of its name (e.g. Resource) followed by the declaration of its fields. A record may extend another record (e.g., Disk extends Resource and in this case Disk is said to be a sub-record of Resource). All the fields of a record are implicitly present in its sub-records. This feature is useful when modeling a type hierarchy such as the class hierarchy in Java. BIR does not impose a restriction that there must a unique top record such as in Java with its `java.lang.Object` class.

BIR supports functions with expression bodies (e.g., IsResourceFree and AreAllResourcesInPoolFreeInv). A function is allowed to be recursive. If recursive calls cause the stack-depth to exceed a user-specified bound, the model-checking process truncates the search along the current path, and a warning is issued indicating the search has not explored all possible states.

A BIR thread contains local variable declarations and a sequence of locations with guarded transformations (transitions) between those locations. A thread with the `main` modifier is the only thread that is active in the initial state of the system. There are two kinds of guarded transformations in BIR: (1) block transformation consisting a sequence of actions that are executed atomically (e.g., `do ...` in `loc0`) and, (2) invocation transformation for function call (e.g. `invoke` in `loc1`). A transformation is enabled when the program counter for the thread is at the location of the transformation, and the transformation's guard holds (if no guard is stated, then the transformation is always enabled when control reaches it). If there are several enabled transformations in a state, then the model checker non-deterministically choose which transformation that will be executed next. If there are no enabled transformation at a state where there are still active threads, then the state is a deadlock state.

In Figure 1, the MAIN thread creates a resource pool with three elements, and starts two Process threads. Each Process thread begins by non-deterministically choosing a resource from the pool. Note that at `loc5`, the process non-deterministically chooses to add the acquired resource back to the pool, or to bypass this action and proceed directly to `loc2`. If the processes keep refusing to relinquish the resources, then eventually the system will reach a deadlock state because the pool is empty (the guard at `loc2` is false). If the second transformation at `loc5` is removed, the system is deadlock free. One can then try to verify the invariant that all resources in the pool should be free (the following subsection explains how Bogor's configuration file is used to specify that the `AreAllResourcesInPoolFreeInv` should be treated as an invariant to be verified).

2.2 Bogor Architecture

Figure 2 presents the Bogor architecture. The architecture of Bogor can be divided into two parts: (1) a front-end that processes a given model expressed in the BIR modeling language, and (2) the actual model checker components. The front-end builds the abstract syntax tree (AST) from the input model, and it then checks the well-formedness of the model, for example, by type checking, and extensions interface checking.

All model-checking tools include functional aspects for state-space search, scheduling, and managing seen before states. However, in their implementations, these aspects are often tangled, and thus insertion of alternate strategies or other customizations is often quite difficult. One of the contributions of our work is not just to present a non-tangled implementation, but moreover to present core components using widely-used and well-documented design patterns [20] that hide irrelevant implementation details by encapsulation, that reduce dependences between components, and that build in strategies for parameterization, adaptation, and extension. Bogor modules need to interact with each other. However, they should only be dependent on the interfaces of other modules. We will describe each Bogor module and the intuition behind its interface, and we will frequently refer to design patterns (or their variations) from [20] using a small caps font, e.g., Abstract Factory. We will only describe the basic capability of each module. Modules can be extended to provide more capabilities via Decorators. A complete listing of the Bogor module APIs and examples of their use can be found on the Bogor web-site [33].

Following the common functional aspects above, the Bogor model checker components consists of three major modules: (1) search module, (2) scheduling module, and (3) state manager module, as well as other modules such as the module that manages backtracking and extensions.

The `ISearcher` is a Strategy for the search method used. For example, if depth-first search is used, then at any given state, its children states will be explored first before exploring its sibling states.

The `IStateManager` is a Facade for managing states. Specifically, the interface dictates that given a state, the

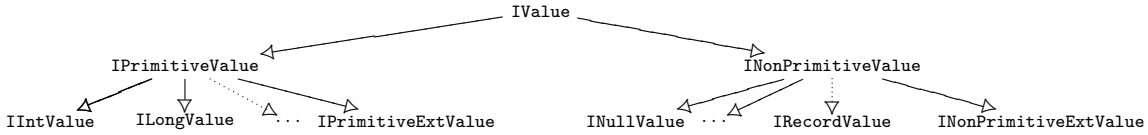

Figure 3: Bogor Value Interface Hierarchy (excerpts)

module determines whether the state has been visited before. There is no constraint on how it achieves that as long as it does not make an assumption that the instance of each state is different. For example, in depth-first search mode there is only one active state at any given time; this is analogous to having one state at runtime. The state is modified by each action and the search relies on the backtracking ability (i.e., undo) to restore the state to the previously visited state. Thus, the module cannot rely on the actual state representation because it changes as program traces are explored.

The `ISchedulingStrategist` is a STRATEGY for the scheduler. The most basic scheduling strategy employed by model-checkers is to generate all possible interleavings of thread executions. Other strategies include incorporation of support for priority based scheduling. In addition, when processing any inner-thread non-deterministic choice (e.g., associated with multiply-enabled transitions within the same thread), this module should be consulted to determine which transition to execute next. For example, in a full-state exploration mode, the scheduler should make sure that every branch of a non-deterministic choice should be explored. This module is also consulted to determine which transformations are enabled in a given state.

The `IActionTaker` is an INTERPRETER for BIR actions such as assignment actions. The module is also a VISITOR for BIR actions, which enables new actions to be incorporated easily if necessary. Given the context of the action (e.g., the state in which it is being interpreted, the thread that executes it, etc.), the module is responsible for executing the action — thus, this module potentially changes the state. For a depth-first search mode, the module is required to produce backtracking information for each action it executes. It is also responsible for managing invocations of all the action extensions that are defined. The `IExpEvaluator` and the `ITransformer` are similar to `IActionTaker` except that they are responsible for executing BIR expressions and transformations, respectively. Furthermore, the `ITransformer` is also responsible for exception handling.

The `IBacktrackingInfoFactory` is an ABSTRACT FACTORY for creating backtracking information for each BIR non-extension action. For example, for a depth-first search, backtracking information consists of method that implements the "undo-ing" of a particular action. The `IStateFactory` and the `IValueFactory` are similar to `IBacktrackingInfoFactory` except that they are responsible for creating BIR states and values, respectively.

Note that all these modules extend the `IModule` interface that governs, for example, how to establish connection between the modules, or how to access each module's options. When an extension is declared in the BIR model, the declaration also includes the fully-qualified name of the Java class that implements it. This Java class should also implement the `IModule` interface as well as all the expression or action extensions that are declared.

Figure 3 presents the hierarchy of BIR values. BIR values are divided into two categories (reflecting BIR types): (1) primitive values, and (2) non-primitive values. When a primitive (non-primitive) type extension is declared, then its corresponding value should implement the `IPrimitiveExtValue` (`INonPrimitiveValue`) interface (this is illustrated in the next section). Non-extension values, their creation, and the determination of their default values are managed by the `IValueFactory` interface. Thus, the specific implementation of the values depends on the implementation of the value factory. However, Bogor requires that a primitive value implementation be immutable, i.e., once created, it cannot be changed. Immutability opens the opportunity for object pooling (via the FLYWEIGHT pattern) whenever appropriate.

A BIR state interface (`IState`) is a FACADE providing the interface to access the global values, thread locations and local values at each of a thread's stack frame.

Bogor is configured through its configuration file. A Bogor configuration defines the implementation of each Bogor module along with their options. For example,

```
ISearcher=myPackage.MySearcher
ISearcher.maxErrors=3
IStateManager=myPackage.MyStateManager
```

specifies that the Java class `myPackage.MySearcher` is the search module, and the `maxErrors` option that specifies the maximum number of errors that the search should try to find before it stops is passed to `myPackage.MySearcher`.

Given a configuration file and a model file, the Bogor front-end will check the well-formedness of the model, and it then instantiates each module (including extension modules) as SINGLETONS. It then wraps them in the `IBogorConfiguration` interface. Each module is then passed the options that are specified in the configuration file, and then the connections between modules are initiated. If there is no error, then the search begins. If the search finds an error, then Bogor produces a counter-example that shows the states along the error trace.

3. SET EXTENSION AS BOGOR PLUG-IN

Figure 4 presents the implementation of the set extension whose interface is defined in Figure 1. `MySet` implements a BIR value that provides the semantics for `Set.type` and the `SetModule` class implements the expression and action extensions declared in `Set`.

In developing extensions, the user has complete control over how to encode the state of the extensions. In the set example, we reuse a Java set implementation (e.g. `java.util.HashSet`) to implement the `MySet`. From another perspective, `MySet` serves as an ADAPTER for the Java set collection library (e.g., the `add` method is an adapter method for `HashSet`). The `linearize` method is a DECORATOR that encodes the state vector of the set value. As with communication layers abstracted by Promela's channel states, we do not need

```
package myPackage;
public class MySet
    implements INonPrimitiveExtValue {
 protected HashSet set = new HashSet();
 protected boolean isNonPrimitiveElement;
 public void add(IValue v) { set.add(v); }
 public byte[][] linearize(...,
    int bitsPerNPV, ObjectIntTable npvIdMap) {
  Object[] elements = set.toArray();
  BitBuffer bb = new BitBuffer();
  if (isNonPrimitiveElement) {
   int[] elementIds = new int[elements.length];
   for (int i = 0; i < elements.length; i++)
    elementIds[i] = npvIdMap.get(elements[i]);
   Arrays.sort(elementIds);
   for (int i = 0; i < elements.length; i++)
    bb.append(elementIds[i], bitsPerNPV);
  } else ...
  return new byte[][] { bb.toByteArray() };
 }
}

public class SetModule implements IModule {
 protected IExpEvaluator ee;
 protected ISchedulingStrategist ss; ...

 public IValue choose(IExtArguments arg) {
  MySet set = (MySet) arg.getArgument(0);
  IValue[] elements = set.elements();
  orderValues(elements);
  int index = ss.advise(..., elements,
        arg.getSchedulingStrategyInfo());
  return elements[index];
 }
 public IValue forAll(IExtArguments arg) {
  String predFunId = ..
  MySet set = (MySet) arg.getArgument(1);
  IValue[] elements = set.elements();
  for (int i = 0; i < elements.length; i++) {
   IValue val = ee.evaluateApply(predFunId,
     new IValue[] { elements[i] });
   if (isFalse(val)) return getBooleanValue(false);
  }
  return getBooleanValue(true);
 }
 public IBacktrackingInfo[]
        add(IExtArguments arg) {
  final MySet set = (MySet) arg.getArgument(0);
  final IValue element = arg.getArgument(1);
  if (!set.contains(element)) {
   set.add(element);
   return new IBacktrackingInfo[] {
       new IBacktrackingInfo() {
         public void backtrack(IState state) {
           set.remove(element);
         } ...
       }
     };
  } else return new IBacktrackingInfo[0];
 }
```

Figure 4: Set Extension Implementation (excerpts)

to hold in the state vector the specific `HashSet` implementation of a set. Rather, the relevant information that we should store is the abstract state of the set, i.e., some compact representation of the members of the set at the current state. The encoding of the members of the set is different for each BIR type. When encoding non-primitive values, Bogor provides a mapping from each non-primitive value to a *unique* id (`npvIdMap`) and to the number of bits needed to encode each id (`bitsPerNPV`). Using this information, the state vector of the set value is a sequence of the unique ids of non-primitive values. In order to preserve set semantics, the ids are ordered, thus, two sets containing the same elements will have the same state vector. The symmetry reduction achieved by storing a canonical representative of a given `HashSet` can significantly reduce model checking costs. The ability to reuse existing Java code in Bogor extensions dramatically reduces implementation costs.

The `SetModule` implements the expression and action extensions in `Set` extension. The `SetModule.choose()` method implements the `choose` expression extension from Figure 1. An expression extension implementation takes an `IExtArguments` and returns an `IValue`. The `IExtArguments` provides the context of the expression evaluation such as the current state and the arguments to the extension expression. The implementation of the `choose` method extracts the set from the argument list, retrieves its elements, and then uses the `ISchedulingStrategist` to select the set element to be returned for the current execution trace. In essence, the `ISchedulingStrategist` is a stateful component that "remembers" what has been previously scheduled at this particular choice point (e.g., it remembers what set elements have been returned to generate previously explored execution traces leading out of this control point). In a full-state exploration, the `ISchedulingStrategist` makes sure that all set elements will eventually be considered. The `forAll()` method applies a predicate to each element in the set. If one element does not satisfy the predicate, then it returns false. The `add()` method adds an element to the set. Since `add` is an action, it must provide backtracking information, i.e., it must provide a method for "undoing" its state transformation. In this example, this is achieved by removing the set element that was inserted by the add operation. Note that each executions of `add` is treated as a single atomic action by the model checker. We believe that analysts should exploit extensions as a means of achieving both an appropriate level of abstraction in modeling and as a means of reducing the state space by controlling atomicity.

As can be seen from this example, implementing extensions does require some knowledge of the internal mechanisms of Bogor. Typically, the most challenging decisions to make are (a) how to implement the linearization function for producing the state vector representation, and (b) how to implement the undo operation required for backtracking. To guide extension developers, we have built a fairly extensive collection of documented examples that illustrate common approaches to obtaining linearization and undo methods. In addition, Bogor provides several facilities to help automate the testing and validation of extensions. In debugging mode, Bogor automatically checks the correctness of undo methods by checking that the state before an action is executed is equivalent to the state that results from executing the action and then immediately backtracking. Furthermore, one can validate various properties of extensions using Bogor by writing and checking small, but general, models. For example, to test our set extension, we created a model that non-deterministically adds and removes elements from a set. After each element addition, we assert that the element is a member of the set, and after each element removal, we assert that the element is not a member of the set.

4. BOGOR APPLICATIONS

In this section we describe our experiences using Bogor to develop customized model checkers for two quite different kinds of software descriptions. The first author, the main developer of Bogor, has implemented a collection of state space encoding and reduction techniques as Bogor plug-ins

that together represent a significant advance in model checking properties of Java programs. This customization of Bogor took approximately 4 weeks, and, over a small range of Java programs, resulted in state space reductions of more than a factor of 20 [15] relative to the best-known techniques [5, 28]. Furthermore, this work has identified additional avenues for further state space reduction as discussed below.

One could argue that this experiences is not representative of the effort that would be required of someone less familiar with Bogor and the implementation of model checkers in general. Perhaps more representative is another experience of a colleague of ours who has a basic knowledge of model checking (but not an in-depth knowledge of the implementation of model checkers or Bogor) and has significant domain knowledge about component-based systems, CORBA middleware, and real-time systems. He started from our existing approach to modeling event-driven CCM-based designs that involved translating them to the input language of the dSpin model checker, an extension of Promela [24]. That translation was developed in over 3 months by a different person. Using the ideas embodied by that translation, he developed, in approximately 1 month, a customization of Bogor that is able to achieve 1000-fold reductions in the size of the state space [14]. We believe this is strong evidence that extension and customization Bogor by domain experts is very cost-effective.

In the rest of this section, we outline the key features of Bogor that were used in these two efforts and then describe how those efforts led to the evolution of Bogor APIs.

4.1 Verification of Java Programs

We have applied Bogor to successfully check real-world size Java programs [15] including a parallel programming framework written in Java that has been used in dozens of real scientific and engineering applications [16]. The task is made easier because BIR can express all features of the Java programming language. The presence of BIR features that directly support, for example, dynamic thread and object creation, exception handling, virtual functions, recursive function, which are lacking in most model checkers, greatly simplified the mapping of Java features. These features of new BIR significantly reduced the cost of implementing the translation from Java to BIR compared to the development of Bandera [11] which translated directly to the older "less featureful" BIR and then on to Promela for the SPIN model checker.

The features of new BIR also make it possible to check a broader range of properties than is possible with the version of the Bandera Specification Language (BSL) [12] that is used in the current distribution version of Bandera. BSL was designed to bridge the gap between a Java-oriented property specification language and the expression and temporal logic language of model checkers like SPIN. The limitations of those expression languages put properties, such as the invariant in Figure 1, beyond the reach of what can be expressed without resorting to the addition of state-space expanding variables. Many complex heap properties, such as, "field `x` only points to an acyclic sorted list where the element contained in each list node is distinct", are not expressible in BSL or any existing model checker input language. Bogor enables these and other complex properties to be expressed without expansion of the state space.

We have implemented various reduction techniques by customizing BIR modules. Specifically, we have implemented heap symmetry [28], thread symmetry [4] reductions, and collapse compression [27] by customizing the `IStateManager` module of Bogor without affecting other modules. We have also implemented a novel partial-order reduction that uses information from a dynamic escape analysis. The idea is that accesses (read/write) to objects that are non-escaping, i.e., reachable only by one thread, do not affect the computation of other threads. We customized the `ISearcher` so that it inspects the heap to determine whether there is an escaping object that will be accessed in the next transition. If that is not the case, then we apply partial-order reduction. This strategy showed upwards of 20 times state-space reduction, over and above the above mentioned reductions, in checking properties of several real Java applications.

4.2 Verification of Cadena Models

Cadena is a design and verification environment for CORBA component model (CCM) systems [24]. CCM systems execute on a CORBA-compliant middleware that provides an event-service (i.e., a framework for implicit invocation). Cadena enriches CCM's component interface definition language (IDL) with specification forms for describing component interconnections, intra-component dependences, and state transition semantics for component methods and event handlers.

While it is possible to produce model checking based analyses of general CCM system designs, we have found that focusing on a specific class of CCM systems allows for significantly more efficient checking tools. Specifically, we focus on the Boeing Bold Stroke architecture which supports the development of real-time component-based embedded systems. Cadena, as described in [24], supports analysis of global behavioral properties of systems by translation to the dSpin model checker. One of the novelties of that approach is the encoding of an abstract semantic model of middleware services in the input language of dSpin. Our Bogor encoding embeds this semantic model in a collection of BIR extensions. Translating Cadena models to this Bold-Stroke-enhanced version of BIR greatly simplifies model development. Furthermore, the BIR extensions separate the static system state (which, because it is not changing, does not need to be held in the state vector) from the mutable state-space. This has a significant impact on memory consumption, and hence performance.

Bold Stroke applications are priority scheduled based on the results of rate monotonic analysis (RMA). In our previous models, we encoded priority scheduling in the dSpin input model, but there were two limitations: additional state variables had to be added to the state space to track the current priority, and for certain compound transitions, e.g., function call and return, priority could not be enforced by our scheme. In Bogor, the input model is completely independent of scheduling information. We simply define a priority assignment for components and then extend the `ISchedulingStrategist` to filter the set of enabled transitions to produce the highest-priority enabled transition. The ease with which the scheduler can be extended in Bogor led us to develop several variant extensions, including an extension of `ISearcher` that keep tracks of an abstraction of the temporal relationship between the period of time-triggered events and the duration of event handlers. The knowledge that drove these extensions was based on observations that are quite natural for a Bold Stroke/CCM domain expert to make, and Bogor provided a mechanism by which they could

be easily incorporated into the analysis.

In addition to customizing parts of Bogor to efficiently reason about Cadena designs, we were able to reuse existing Bogor components. Some of these, for example Bogor's collapse compression component, provide a significant reduction over what could be obtained with existing model checkers. Preliminary performance data suggests that this customization allows for orders of magnitude reductions in the state-space [14] over what could be achieved using translation approach [24]. While it is possible to modify the implementation of dSpin to achieve similar results, this would require gaining in-depth knowledge of dSpin's implementation which is a significant effort. We believe this experience demonstrates that customizing Bogor produced what amounts to a dedicated model checker for Cadena, but without paying the cost of implementing it from the scratch.

4.3 The Evolution of APIs

We do not claim that Bogor has the ideal collection of APIs. By designing for extension, we hope that the current set of APIs will admit a wide variety of uses for Bogor, but we expect that the APIs will evolve as users gain experience in customizing Bogor. In the two projects described above we identified a common form of reduction that was extremely useful; we call it *dynamic atomicity*. Most model checker languages allow a fragment of the input to be marked as *atomic* and that fragment will be treated as a single indivisible transition in state space exploration. This static decision of what should be considered atomic is a limitation. A more general approach is to define a predicate over a system state and a transition that when true allows the atomic execution of the transition. The customizations of Bogor described above both use this idea but each of them had to implement the mechanism that applies the predicate and then drives the atomic transition execution. We are redesigning Bogor's APIs to factor this common functionality and to require only the predicate definition in order to achieve dynamic atomicity reductions.

5. RELATED WORK

There exists a large body of work on tools for finite-state verification of software. The Spin [26] model checker was initially designed to verify communication protocols. However, it has been succesfully applied for other software artifacts such as Java programs, for example, as the back-end of the Bandera project [11]. Similarly, the NuSMV [9] model checker is designed for checking hardware, but it has also been used in the Bandera project to verify Java programs. Both model-checkers have been widely used due to their level of maturity and efficiency. However, they lack some features essential for checking modern, dynamic, and concurrent software such as Java programs. When using them with Bandera, we resorted to bounded-static approximations of dynamic behaviors of the programs. In addition, because they lack of features such as dynamic heap object creation and management, many reductions strategies [28, 15] that have proven to be useful for verifying dynamic software cannot be easily incorporated. Furthermore, the designs of the tools are not flexible and their internal functional aspects are not decoupled enough to enable customization without knowing the details of how the tools are implemented.

The SLAM [2], BLAST [25], and MAGIC [7] model checkers work on "well-behaved" (e.g., no pointer arithmetic) sequential C programs with minimal support for dynamically allocated structures. Although these tools are similar to Bandera in that they support the checking of source code, the research programs for these projects are fairly different since they emphasize support for automatic abstraction based on counter-example driven predicate abstraction refinement. In contrast, the research program for Bandera has focused on support checking of highly-dynamic concurrent object-oriented source code. None of the tools mentioned above has an emphasis on extensibility nor do they have a goal of checking a variety of types of software artifacts.

The closest projects to ours are dSpin [13] and JPF [5]. Iosif's dSpin model checker is a version of Spin extended with features such as dynamic creation of heap objects and garbage collection, and support for more features such as exception handling is under development. The significant contribution of dSpin is its ground-breaking approach to canonical heap representation upon which Bogor's heap representation is based. Customizing dSpin is fairly difficult since it is built on top of an old non-modular code-base (the implementation of Spin) and since it takes an indirect generative approach to model-checker implementation. (i.e., given a Promela specification, it produces the C code for a model-checker dedicated to the given specification). When using dSpin for checking Cadena models, because it was an overwhelming challenge to modify the internals of dSpin, we had to resort to explicitly guarding each transition with priorities in the model and to adding extra state variables to hold the priority-based scheduling information. In addition, we had to model the CORBA event-service directly using dSpin basic constructs instead of hiding portions of the state in extensions. Using Bogor results in much greater efficiency because we are able to easily customize the scheduling and search modules, and because we introduce abstract constructs that are specific to CORBA.

JPF was ground-breaking in the area of software model-checking in that it demonstrated that a model-checker can be built to work directly on a very semantically rich language like Java byte-code. JPF is well-designed, and its flexibility has been demonstrated by the incorporation of a variety of search modes such as heuristic searches [23, 32]. The direct support for Java features and the flexibility of JPF has been a significant source of inspiration for our work on Bogor. In fact, Willem Visser, the primary developer of JPF was the person who originally suggested that we should write a model-checker to work directly on BIR. In some settings, the fact that JPF works directly on Java byte-code is an advantage: `javac` implements the translation to the model-checker input language, and one can claim to some extent that the code being checked is the code that is actually going to be run (modulo differences in virtual machines). However, in many applications that we wish to support, working directly on byte-codes is a disadvantage. Our goals with Bogor are to provide an extensible modeling language that can support checking of artifacts at different levels of abstraction. Thus, Bogor offers a compromise: it supports checking of Java, and yet, it is flexible enough to be adapted to obtain dedicated checkers for different families of software artifacts.

6. CONCLUSIONS

As model checking continues to grow in popularity as a software analysis framework, model checking tools need to adapt so that they will effectively support a broad range of system descriptions and property languages. One approach to overcoming the significant cost of model checking is to

exploit available domain knowledge of specific software artifacts to develop highly-optimized state space representations, reductions and search algorithms. We believe that Bogor's extensible, customizable tool architecture will help minimize the amount of model checking-specific knowledge that a domain expert will need to build cost-effective analysis capabilities.

We are making Bogor publicly available,[1] along with extensive API documentation and examples of plug-ins and extensions, as a means of catalyzing the use of model checking in the broader software analysis community. Up to date information on Bogor is available at [33].

7. REFERENCES

[1] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proceedings of the 6th European Software Engineering Conference*, pages 175–188, Nov. 1998.

[2] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 203–213, June 2001.

[3] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering: An International Journal*, 6(1):37–68, Jan. 1999.

[4] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *International Journal on Software Tools for Technology Transfer*. Springer-Verlag, 2002.

[5] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.

[6] T. Bultan, R. Gerber, and C. League. Composite model-checking: verification with type-specific symbolic representations. *ACM Transactions on Software Engineering and Methodology*, 9(1):3–50, Jan. 2000.

[7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *Proceedings of the 25th International Conference on Software Engineering (to appear)*, 2003.

[8] W. Chan, R. J. Anderson, P. Beame, D. Notkin, D. H. Jones, and W. E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Transactions on Software Engineering*, 27(2):170–190, 2001.

[9] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[11] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera Specification Language. *International Journal on Software Tools for Technology Transfer*, 2002.

[13] C. Demartini, R. Iosif, and R. Sisto. dspin : A dynamic extension of SPIN. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, Sept. 1999.

[14] W. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh. Model-checking middleware-based event-driven real-time embedded software. Technical report, Kansas State University, 2003. (submitted for publication).

[15] M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Using static and dynamic escape analysis to enable model reductions in model-checking concurrent object-oriented programs. Technical report, Kansas State University, 2003. (submitted for publication).

[16] M. B. Dwyer and V. Wallentine. A framework for parallel adaptive grid simulations. *Concurrency : Practice and Experience*, 9(11):1293–1310, Nov. 1997.

[17] Eclipse Consortium. Eclipse website. `http://www.eclipse.org`, 2001.

[18] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with hsf-spin. In *Model Checking Software, Proceedings of the 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, May 2001.

[19] FormalSystems. FDR2 website. `http://www.fsel.com/`, 2003.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub. Co., 1995.

[21] D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (to appear)*, May 2003.

[22] P. Godefroid. Model-checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 174–186, Jan. 1997.

[23] A. Groce and W. Visser. Model checking java programs using structural heuristics. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 12–21. ACM Press, 2002.

[24] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering (to appear)*, 2003.

[25] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.

[26] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[27] G. J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, Apr. 1997.

[28] R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, Apr. 2002.

[29] C. T. Karamanolis, D. Giannakopolou, J. Magee, and S. M. Wheather. Model checking of workflow schemas. In *Proceedings of the 4th International Enterprise Distributed Object Computing Conference*, pages 170–181, Sept. 2000.

[30] N. Kaveh. Model checking distributed objects design. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[31] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing MHP information for concurrent java program. In *Proceedings of the 7th European Software Engineering Conference*, pages 338–354, Sept. 1999.

[32] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, Apr. 2001.

[33] Robby, M. B. Dwyer, and J. Hatcliff. Bogor Website. `http://www.cis.ksu.edu/bandera/bogor`, 2003.

[1] An initial release is scheduled for the Summer of 2003.